Pharo Morphic Application Framework

skrishnamachari@gmail.com

Abstract

The purpose of this framework is to have a comprehensive Pharo Application hierarchy to easily construct an enterprise ready application, that is scalable and built out as modules that link together.

Morphic View Framework: Simplify creating Morphic UI with creating views from

Components/ Parts / Modular UI construction: Create multiple pieces of independent views and compose it into one composite application view easily with:

- Specs based View Construction
- Simple Morphic View
- Panels Composition: Complex GUI
- API for: View attributes initializations intervention in intermediate steps
- SupervisoryController aka MVP Presenter
- Locale based translation integrated
- LoggerManager to easily plugin Toothpick logger
- Configuration based constants from external properties file / defaults code based..
- Named Prototype to optimize space/ time complexities as well as extension to dependency injection
- Keyboard Event Handler at view level
- Mouse Listener at a view level
- Flexible Explicit View Construction

Additionally incorporate existing/create standard frameworks, possibly as plugins or independent package, for:

- Evolve a UI Builder that is a simple drag n drop capability of widgets
- Complete RPC capabilites over XMLRPC for various purposes vis:
  - a JDBC connector along with an example complimentary Groovy Server backend / Groovy Client code

References:

1. Various Smalltalk implementations: MVC architecture leading to the SupervisoryController architecture of MVP.
2. Spring Framework
3. General Enterprise Application Architecture patterns

Motivation:

Smalltalk is a beautiful and enjoyable language. Morphic is a wonderful but most under exploited UI framework that has immense potential to exponentially grow with contribution from the world of developers.

Enterprise developers today are not experts, but casual programmers who depend on existing frameworks, patterns to adapt quickly. The large scale programming will require the ability to cleanly modularize and work in silo's but in one coherent architectural framework. This framework should cater to the now evolving standard features and capabilites in enterprise applications as stated in the abstract above.

Also important is the ability of any framework to easily to talk to all of the existing IT assets, be it built in Smalltalk, Java, .Net , C++, Python or Ruby in failsafe manner and easily. Performance be acceptable to begin with and optimizable.

The framework should extend its reach between standard large scale enterprise application as well as the now fast evolving tablet marketplace based application delivery that are small atomic units of apps.

There are great potentials that Pharo Smalltalk can quickly evolve to exploit before the market fully saturates

Get Started:

- Load the http://www.squeaksource.com/PharoGoodies: ***ConfigurationOfPharoMorphicViewCoreApp*** latest : skr.4 version from the http://www.squeaksource.com/PharoGoodies

Quick Introduction:

Let us see a simplest of Morphic View construction to get started with:

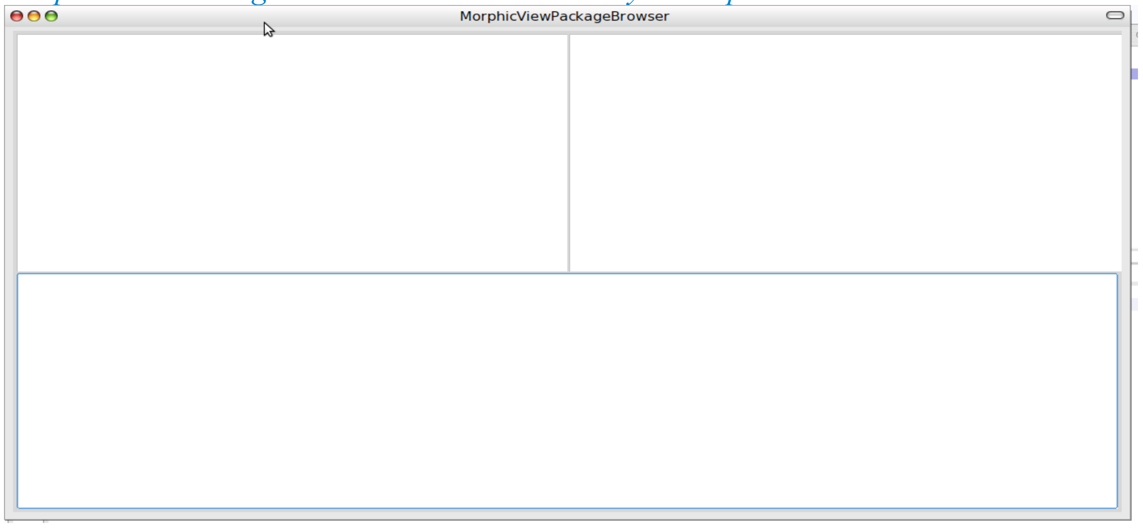
subclass AbstractMorphicView and implement:
#createMorphsSpecs , #layoutSpecsArray, #initialize

Refer to the implementation code at: **Code Hyper Cards:** MorphicViewPackageBrowser Basic 01

Thats it, you have the basic prototype ready...

Use any of the following to check it out..from a workspace

MorphicViewPackageBrowser open .
OR: MorphicViewPackageBrowser new open.
OR: MorphicViewPackageBrowser new createPrimaryView openCenteredInWorld



Add functionality to this prototype:

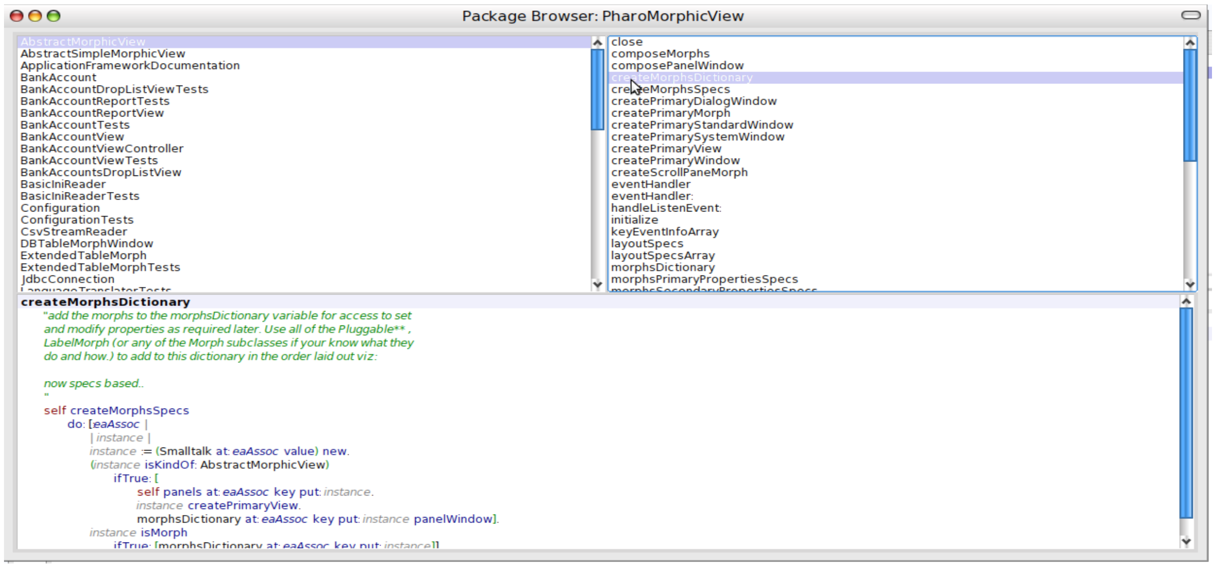
To flesh out the application to a minimal complete capability:

Implement #**morphsPrimaryPropertiesSpecs , #morphsSecondaryPropertiesSpecs**
modify: #**initialize**

Add a new model class: **MorphicViewBrowserModel** with instance variables:
classList, classSelectionIndex, methodSelectionIndex

create accessors for all the variables
and add methods for: **#methodList , #methodSource**

Fill out the methods as per listing in **Code Hyper Cards**: MorphicViewPackageBrowser 02
Now this should have the basic code browser up and running:
(MorphicViewPackageBrowser new packageName: 'PharoMorphicView') open.



```
AbstractMorphicView
AbstractSimpleMorphicView
ApplicationFrameworkDocumentation
BankAccount
BankAccountDropListViewTests
BankAccountReportTests
BankAccountReportView
BankAccountTests
BankAccountView
BankAccountViewController
BankAccountViewTests
BankAccountsDropListView
BasicIniReader
BasicIniReaderTests
Configuration
ConfigurationTests
CsvStreamReader
DBTableMorphWindow
ExtendedTableMorph
ExtendedTableMorphTests
JdbcConnection
```

```
close
composeMorphs
composePanelWindow
createMorphsDictionary
createMorphsSpecs
createPrimaryDialogWindow
createPrimaryMorph
createPrimaryStandardWindow
createPrimarySystemWindow
createPrimaryView
createPrimaryWindow
createScrollPaneMorph
eventHandler
eventHandler:
handleListenEvent:
initialize
keyEventInfoArray
layoutSpecs
layoutSpecsArray
morphsDictionary
morphsPrimaryPropertiesSpecs
```

```
createMorphsDictionary
    "add the morphs to the morphsDictionary variable for access to set
    and modify properties as required later. Use all of the Pluggable** ,
    LabelMorph (or any of the Morph subclasses if your know what they
    do and how.) to add to this dictionary in the order laid out viz:

    now specs based..
    "
    self createMorphsSpecs
        do: [:eaAssoc |
            | instance |
            instance := (Smalltalk at: eaAssoc value) new.
            (instance isKindOf: AbstractMorphicView)
                ifTrue: [
                    self panels at: eaAssoc key put: instance.
                    instance createPrimaryView.
                    morphsDictionary at: eaAssoc key put: instance panelWindow].
            instance isMorph
                ifTrue: [morphsDictionary at: eaAssoc key put: instance]]
```

You can play with the little refactoring done for formatted text display, classSelected, methodSelected methods. Extended package with labels, buttons, menu and keyboard shortcuts, drag/drop event will be natural test of the entire advanced framework.

Advanced MorphicView Application Framework

For a more complete framework , with extended functionality:

- Load the http://www.squeaksource.com/PharoGoodies:
  ConfigurationOfPharoMorphicView latest : skr.9 version
  from the http://www.squeaksource.com/PharoGoodies

All examples and all other Tests, Examples packages will take you through the extensions of the example to utilize the current framework capabilities in:

Simple Morphic View:

Allows construction of the view with Tabular layout using #column layout as default to obtain simplest of UI's without even the #layoutSpecsArray being described with just their extents being specified. (There are bugs in PluggableButtonMorph not honoring extents if #label: is called, resising to the label extent).

Check out on the TestNewSimpleMorphicFromSpecsView from the Tests package.

Change the panelType: #row to compose row based UI and even compose a complex UI with panels of row/ column composed views as subclasses of AbstractSimpleMorphicView class.

Panels Composition:

The idea of a complex GUI is when you can compose a View with other view panels integrated in the main panel to any depth / levels required.

Its simple to do so in:

```
#createMorphsSpecs
        ^{
        'panelView1' -> #ViewClassName.
        ...
        }
```

rest of the methods will flow as usual with actions/ attributes to be set IF required in the morphsPrimaryPropertiesSpecs or morphsSecondaryPropertiesSpecs for this widget.

View attributes initializations:

The framework supports initializations of properties at various steps of the construction including the #initialize and #postOpen that is before and after this method call:

```
AbstractMorphicView >>
createPrimaryView
"the primary builder method"
        self createMorphsDictionary.
        self panelViewInitialize.
        self setMorphsPrimaryProperties.
        self setMorphsSecondaryProperties.
        self layoutSpecs.
        self createPrimaryWindow.
        self composePanelWindow.
        self postCreateInitialize.
        self composeMorphs.
        ^ primaryWindow
```

Override these methods at appropriate stage for setting the attribute values for your application.

SupervisoryController aka MVP Presenter:

MorphicViewPackageBrowser

```
    initialize
    ...
    self viewController: MorphicViewBrowserController new.
    self  viewModel: MorphicViewBrowser new.
    ...
```

This is the MVP SupervisoryController for actions on the View in:

```
    morphsPrimaryPropertiesSpecs
    ^{
    ...
    'instanceMethodsBTestNewSimpleMorphicFromSpecsView
TestNewSimpleMorphicViewutton' -> { #on: -> { viewController .  } }.
    'classMethodsButton' -> { #on: -> { viewController .  } }.
    }
```

similar adaptation of methods in #**morphsSecondaryPropertiesSpecs** can be done to set any property for the widget

Locale Translation:

for any strings to be simply translated based on the Locale just add *getLocaleString* after the string viz:

```
    morphsSecondaryPropertiesSpecs
    ^{
    ....
    'instanceMethodsButton' -> {
        #label: -> 'instance methods' getLocaleString }.
    ...
    }
```

The translation shall be picked up if you provide a translation csv file of two columns mapping the existing strings in the application and the translated string. Create: en-US_PharoNLS.csv or fr-FR_PharoNLS.csv / others as reqd, in the current working directory. Auto generation of this file will be added to this framework later.

LoggerManager

Currently LoggerManager uses the Toothpick framework and simplifies its integration with simple steps of:

Initialize with in the startUp class side method:

startUp
PharoMorphicObject logger .
logger class loggersStart: #(transcript file stream).

then add either of these message anywhere in your code:

logger log: 'logging messages' .
logger log: 'logging message' level: #debug.
logger log: 'logging message' category: #default level: #debug.

Configuration based guiConstants

class side methods:

guiConstants

```
^super guiConstants
	at: 'primaryWindowColor' ifAbsentPut: (Color darkGray alpha: 0.9);
	yourself
```

These are primarily based on a principle of Configuration file defining the values and if missing , provided with the defaults in the class side.

The Configuration filename is defined in *Configuration class>> configurationFiles* as application_properties.ini . This is in an ini format of key value pairs in the base working directory.

The configuration are read and cached and you will have to reset it for re-reading the changes to the ini file.

Usage is simple (*self class guiConstants at: 'key'*) in any instance method to assign/ use the value

Named Prototype

This is a small framework piece with a huge utility and implications that can be both beneficial and quite complex and throw contortions if misused. (reset NamedPrototype instance whenever in doubt and check your logic, it will help eliminate inherent inefficiencies applications posses)

The framework provides for a default weak cached dictionary that holds the instances created and can thereafter be obtained in any method with its naming. This is akin to a prototype in Self, but not as aggressively wired in. There are huge possibilities of its utility as explored in the Java Spring Framework. Simplest is its utility to avoid creating multiple instances when a defined set can be reused any number of times, for singletons that can later extend, prototypes with veryDeepCopy providing for easiest recreation of an instance copy.

If the class is in the hierarchy of PharoMorphicObject.

aBrowser := MorphicViewPackageBrowser named: 'Examples.PackageBrowser' .
aBrowser open.

This now guarantees it to be a singleton whereever invoked.
aModel := MorphicViewPackageBrowserModel named: 'Examples.PackageBrowserModel'.
aModel classList: Smalltalk allClasses.
aModel02 := NamedPrototype instance copyFrom: 'Examples.PackageBrowserModel' to: 'Examples.PackageBrowserModel02'

this creates a deep copy of the model that is used as a prototype.

Keyboard Event Handler

add:

MorphicViewPackageBrowser

initialize
...
self eventHandler: #MorphicViewBrowserEventHandler.
...

keyEventInfoArray
^ {
{64. $O. viewController . #openPackage}.
{8. $M. viewController . #openImplementors} .
{8. $N. viewController . #openSendors} .
}

the methods be implemented in the Controller class for their appropriate actions

Mouse Listener:

add:

MorphicViewPackageBrowser

handleListenEvent: anEvent
anEvent type == #dropEvent ifTrue:[^viewController handleDropEvent: anEvent].

Implement the handleDropEvent: method in the controller.

Complex mouse handling can be done with initializing mouseListener and the eventHandler implements mouseDown: / mouseOver: etc... Refer to examples for details

Flexible Explicit View Construction:

The view framework is not tightly bound to building the view up from specs only. You can have the exact same framework mix n build with explicit widget instantiation and more scalable compositions. Specs can have limitations in what they can instantiate and will require explicit widget build out capabilities to mesh in.

Override the methods optionally of:

#createMorphsDictionary , #setMorphsPrimaryProperties , #setMorphsSecondaryProperties

Refer to the examples and Tests for this capability fully demonstrated.

the use of specs and explicit methods are completely developer's choice and supported, though the understanding of the overridden super methods is mandatory and recommend knowing the details of what they achieve to override appropriately.

Review the current tests/ sample implementation of:

Package: PharoMorphicView-Tests
"Simplest test implementations of the 4 alternative permutations"

- TestNewProportionalMorphicFromSpecsView
- TestNewProportionalMorphicView
- TestNewSimpleMorphicFromSpecsView
- TestNewSimpleMorphicView

Package: PharoMorphicView-Examples-SurabhiMathsLesson
"A simple Abacus implementation with a good example of custom spec creation in code to build out a simple view with configurable number of widgets in a defined pattern of layout, behaviour. Simple Games/ Children lessons can find parallels here in this use of the framework"

Workspace Text:

```
SurabhiMathLessonView new open.
```

** The Framework details will be expanded on in the next version of this guide, with more details of the ProportionalLayouts, TableLayouts and the Morphic best practices being uncovered.

Pharo Code Hypercard

(print and refer)

MorphicViewPackageBrowser Basic 01

MorphicViewPackageBrowser

```
        AbstractMorphicView subclass: #MorphicViewPackageBrowser
                instanceVariableNames: ''
                classVariableNames: ''
                poolDictionaries: ''
                category: 'PharoMorphicView-Examples-PackageBrowser'
```

Add instance methods:

createMorphsSpecs

```
"The widgets you want in the View"
        ^{
        'classList' -> #PluggableListMorph.
        'methodsList' -> #PluggableListMorph.
        'methodSource' -> #PluggableTextMorph.
        }
```

layoutSpecsArray

```
"The layout frame spec for each widget"
        ^{
        'classList' -> {
                        #fractions -> (0 @ 0 corner: 0.5 @ 0.5).
                        #offsets -> ( 4 @ 4 corner: 0 @ 0).
                        }.
        'methodsList' -> {
                        #fractions -> (0.5 @ 0 corner: 1 @ 0.5).
                        #offsets -> (2 @ 4 corner: 0 @ 0).
                        }.
        'methodSource' -> {
                        #fractions -> (0 @ 0.5 corner: 1 @ 1).
                        #offsets -> ( 4 @ 2 corner: (-4 @ -4)).
                        }.
                }
```

initialize

```
"minimal to start with "
        super initialize.
        windowType := #Window.
```

Workspace Text to run:

```
        MorphicViewPackageBrowser new  open.
```

Pharo Code Hypercard

(print and refer)

MorphicViewPackageBrowser 02

Carry forward from the earlier code: additional instance methods on **MorphicViewPackageBrowser**

morphsPrimaryPropertiesSpecs

```
" add the primary actions on the widgets"
	super morphsPrimaryPropertiesSpecs.
		^{
	'classList' -> ( #on:list:selected:changeSelected:menu:keystroke: -> { viewModel. #classList. #classSelectionIndex . #classSelectionIndex: . nil. nil } ).
	'methodsList' ->  ( #on:list:selected:changeSelected:menu:keystroke: -> { viewModel. #methodList. #methodSelectionIndex . #methodSelectionIndex: . nil. nil } ).
	 'methodSource' -> ( #on:text:accept:readSelection:menu: -> { viewModel. #methodSource . #methodSource: . nil . nil } ).
	}
```

morphsSecondaryPropertiesSpecs

```
"just an example"
	super morphsSecondaryPropertiesSpecs.
	^{
	 'methodSource' ->{
	 #font: -> { StandardFonts codeFont} .	}.
	}
```

now modify the #intialize method:

initialize

```
	super initialize.
	windowType := #Window.
	self viewModel: MorphicViewBrowserModel new.
	viewModel addDependent: self.
```

add:

packageName: aPackageName

```
	viewModel classList: (PackageOrganizer default packageNamed: packageName ) classes
```

windowTitle

```
	^'Package Browser: '
```

Create the model class:

```
MorphicViewController subclass: #MorphicViewBrowserModel
	instanceVariableNames: 'classSelectionIndex classList methodSelectionIndex'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'PharoMorphicView-Examples-PackageBrowser'
```

create accessors for the instance variables:
classList ; classSelectionIndex ; methodSelectionIndex
initialize

```
  classList := Smalltalk allClasses.
  classSelectionIndex := 1.
  methodSelectionIndex := 1.
```

methodList

```
        ^( (self classList at: self classSelectionIndex ) methods
                                        collect: [:ea | ea selector]) asSortedCollection.
```

methodSource

```
        ^(self classList at: self classSelectionIndex ) sourceCodeAt: (self methodList at: self
methodSelectionIndex)
```

methodSource: aText

```
"no – op for now"
```

now modify these accessors:

classSelectionIndex: anIndex

```
        classSelectionIndex := anIndex.
        methodSelectionIndex := 1.
        self changed: #classSelectionIndex.  "the update: of the pluggable list morph will be called"
        self changed: #methodList.
```

methodSelectionIndex: anIndex

```
        methodSelectionIndex := anIndex.
        self changed: #methodSelectionIndex.
        self changed: #methodSource
```

Workspace Text to run:

```
        (MorphicViewPackageBrowser new packageName: 'PharoMorphicView' ) open.
```

** Additional Code Cards shall be brought in the next version

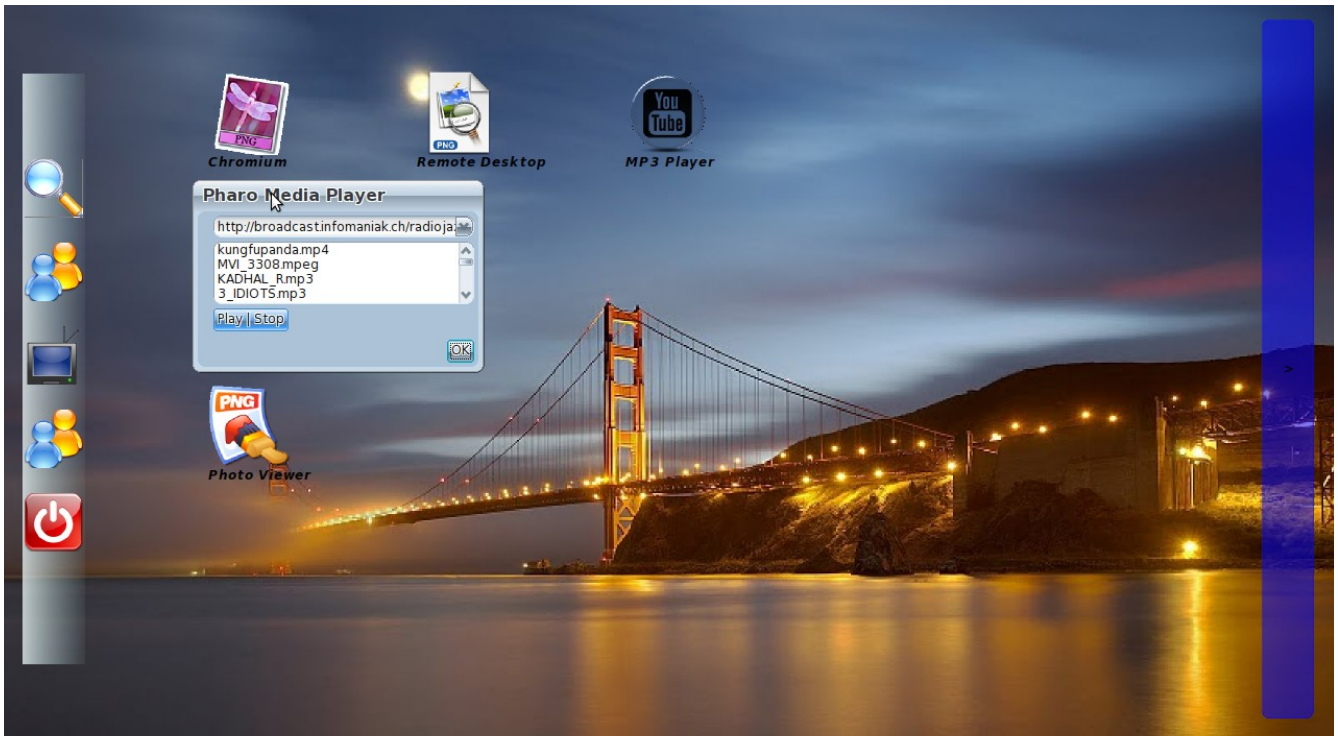
APPENDIX

Framework will progress to provide, incorporate existing/create standard frameworks, possibly as plugins, for:

- Morphic best practices are embedded in the framework, eliminating suggested deprecations of classes/ methods
- Tablet based IDE that is seamless in appearance on the desktop, laptop or a tablet and perhaps the standard phone or any device alike.
- Tabular Data / DataRecord / DataSet / Lazy load patterns that is distinct from the GLORP ORM based on customizable sql transactions
- Integrate other enterprise application architectural patterns relevant to this framework for easy reuse/ extensions
- RPC Framework capabilites over XMLRPC for various purposes viz:
    - Secure layer along with simple protocol to obtain optimized XMLRPC data transfer
    - JDBC connector along with an complimentary Groovy Server backend code
    - Jasper Reports
    - Other capabilities one can directly exploit of available frameworks from Groovy/ Python/ Ruby backends with minimal build out of their server side components
    - Capability to call in Pharo from outside for any queries, reporting etc..
- Flexible and simple enough to plugin existing capabilities of GLORP to feed from the models of this architecture without modifications, but additional minimal code.
- External output in various formats of XML, JSON, CSV from application model(s) easily with a markup builder
- Integrate a basic Rules Engine built few years back to provide for externalized configurable file/ DB based business logic for the application to ease scalability/ flexibility in maintaining the application.

Preview of a Runtime Pharo Application World:

This will be the target of a complete runtime multi panel World in the lines of the standard tablet IDE's with contributory applications appear on screen automatically/ installable into.

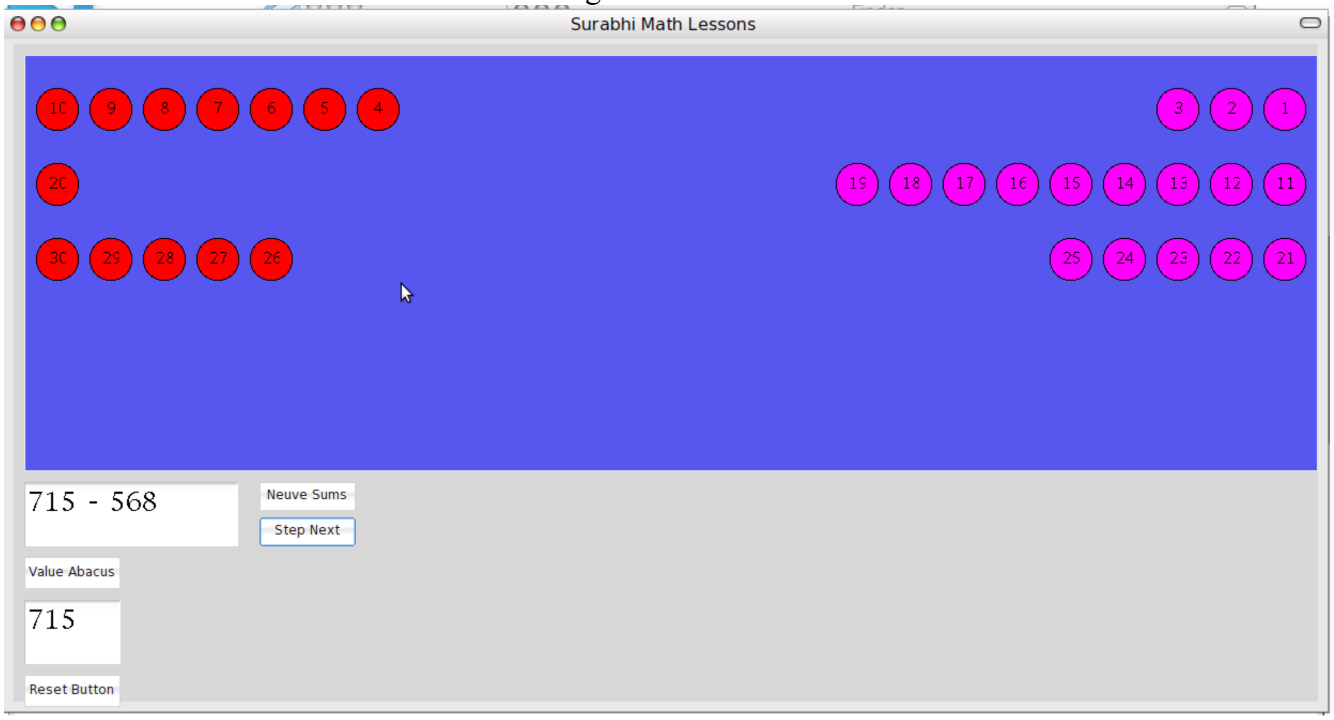


This is an ambitious attempt to have the capabilities of a tablet IDE deliverable to all targets of desktop/laptop, tablets on the current target OS of Windows, Linux and on a Tiny Core linux for tablets and Android maybe in future

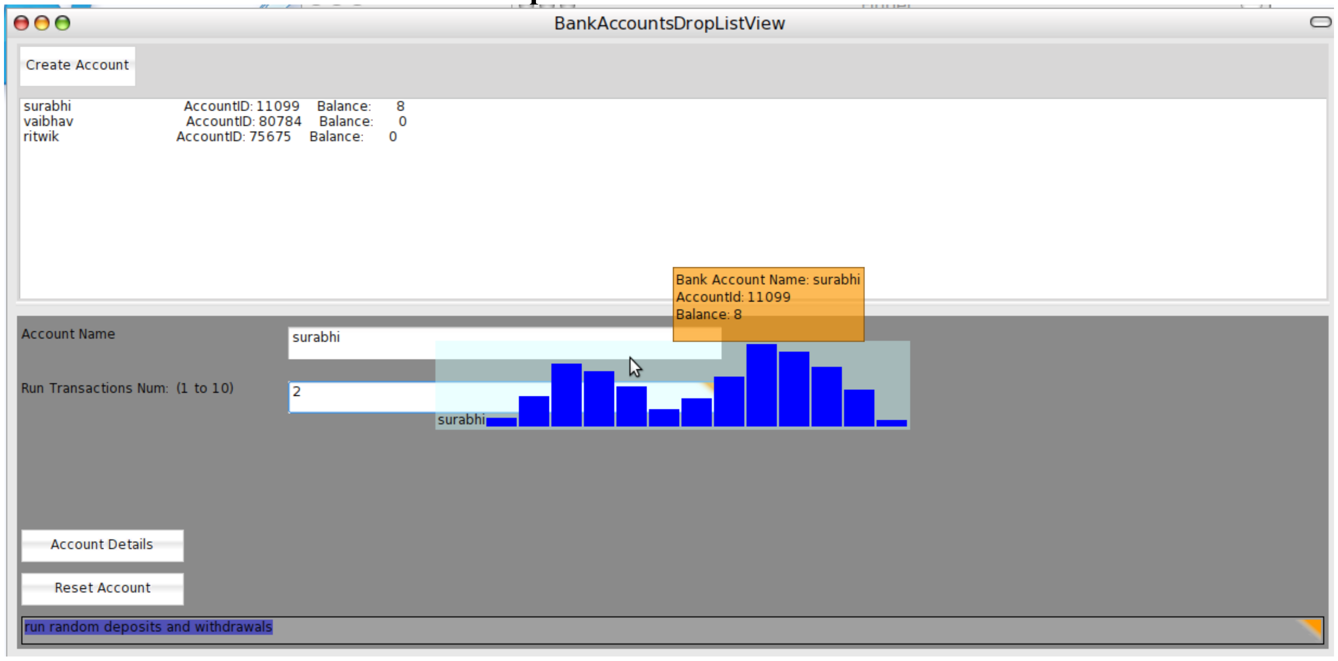
Screenshots

(some of the applications built out with the framework)

Abacus: Configurable to number of lines..



Sample Bank Accounts View:



```
surabhi          AccountID: 11099   Balance:   8
vaibhav          AccountID: 80784   Balance:   0
ritwik           AccountID: 75675   Balance:   0
```

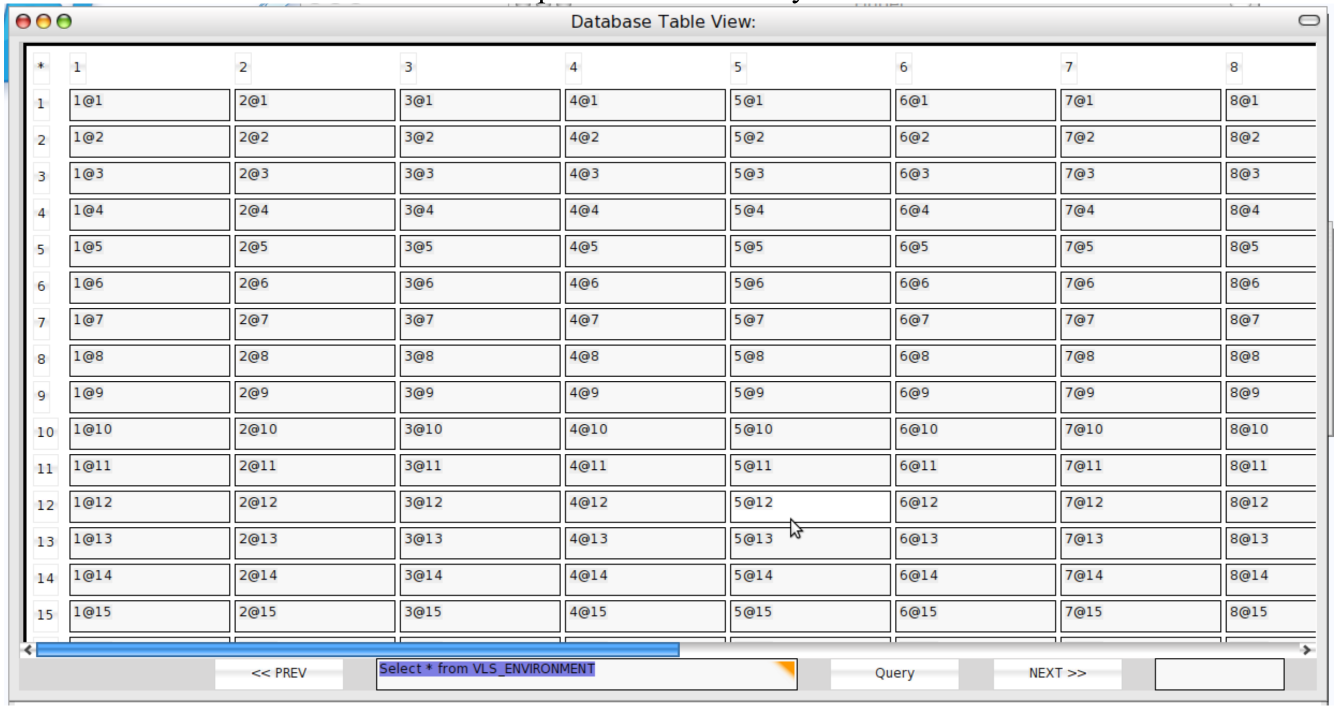
Bank Account Name: surabhi
AccountId: 11099
Balance: 8

Account Name: surabhi

Run Transactions Num: (1 to 10): 2

run random deposits and withdrawals

Sample Tabular Data Layout:



Database Table View:

*	1	2	3	4	5	6	7	8
1	1@1	2@1	3@1	4@1	5@1	6@1	7@1	8@1
2	1@2	2@2	3@2	4@2	5@2	6@2	7@2	8@2
3	1@3	2@3	3@3	4@3	5@3	6@3	7@3	8@3
4	1@4	2@4	3@4	4@4	5@4	6@4	7@4	8@4
5	1@5	2@5	3@5	4@5	5@5	6@5	7@5	8@5
6	1@6	2@6	3@6	4@6	5@6	6@6	7@6	8@6
7	1@7	2@7	3@7	4@7	5@7	6@7	7@7	8@7
8	1@8	2@8	3@8	4@8	5@8	6@8	7@8	8@8
9	1@9	2@9	3@9	4@9	5@9	6@9	7@9	8@9
10	1@10	2@10	3@10	4@10	5@10	6@10	7@10	8@10
11	1@11	2@11	3@11	4@11	5@11	6@11	7@11	8@11
12	1@12	2@12	3@12	4@12	5@12	6@12	7@12	8@12
13	1@13	2@13	3@13	4@13	5@13	6@13	7@13	8@13
14	1@14	2@14	3@14	4@14	5@14	6@14	7@14	8@14
15	1@15	2@15	3@15	4@15	5@15	6@15	7@15	8@15

Menubar addition from the framework:

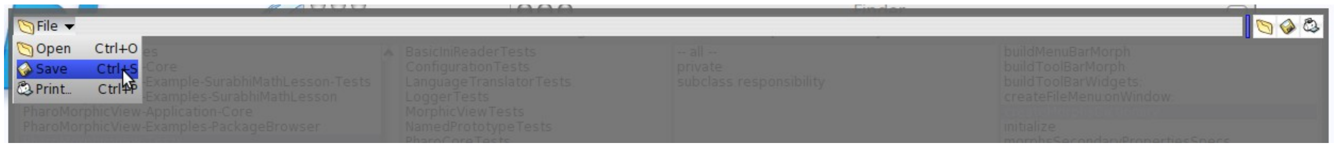


Table of morphs: Standalone widget adapted to work with array of objects